Bluespec-7: Scheduling & Rule Composition

Arvind
Computer Science & Artificial Intelligence Lab
Massachusetts Institute of Technology

GAA Execution model

Repeatedly:

- Select a rule to execute
- Compute the state updates
- Make the state updates

Highly non-deterministic

User annotations can help in rule selection

Implementation concern: Schedule multiple rules concurrently without violating one-rule-at-a-time semantics

Rule: As a State Transformer

A rule may be decomposed into two parts $\pi(s)$ and $\delta(s)$ such that

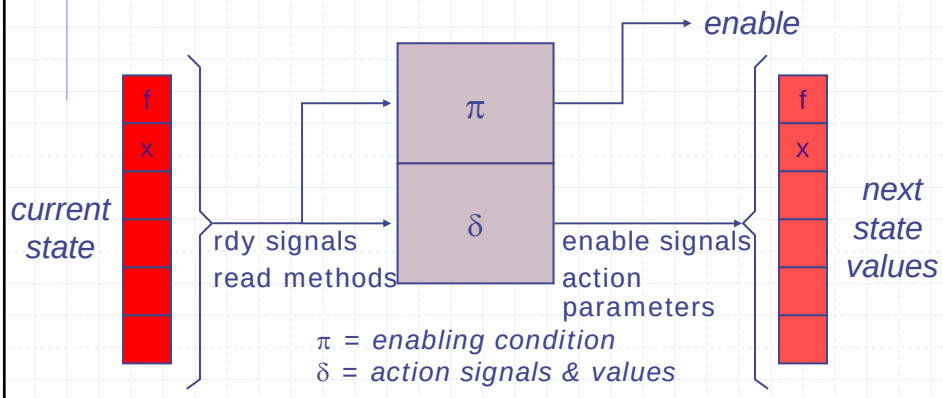
$$s_{next} = \textit{if } \pi(s) \textit{ then } \delta(s) \textit{ else } s$$

$\pi(s)$ is the condition (predicate) of the rule, a.k.a. the "CAN_FIRE" signal of the rule.
(conjunction of explicit and implicit conditions)

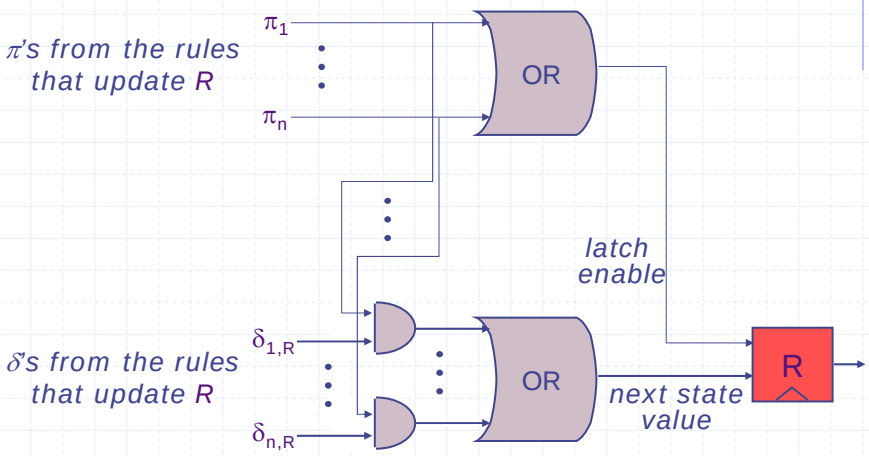
$\delta(s)$ is the "state transformation" function, i.e., computes the next-state value in terms of the current state values.

Compiling a Rule

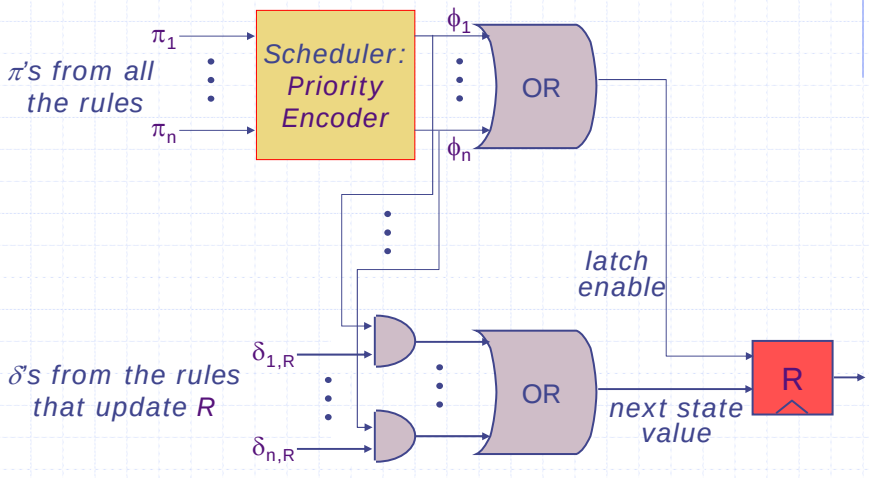
```
rule r (f.first() > 0) ;
        x <= x + 1 ;   f.deq ();
endrule
```



Combining State Updates: *strawman*

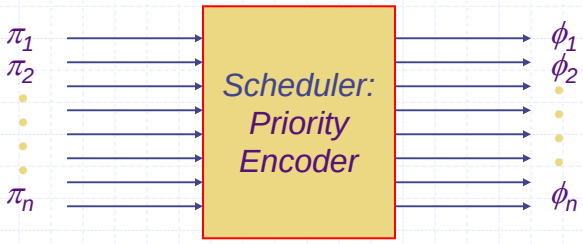


Combining State Updates



Scheduler ensures that at most one ϕ_i is true

One-rule-at-a-time Scheduler



1. $\phi_i \Rightarrow \pi_i$

2. $\pi_1 \vee \pi_2 \vee .... \vee \pi_n \Rightarrow \phi_1 \vee \phi_2 \vee .... \vee \phi_n$

3. One rewrite at a time
   i.e. at most one ϕ_i is true

Very conservative way of guaranteeing correctness

Executing Multiple Rules Per Cycle: *Conflict-free rules*

```
rule ra (z > 10);
   x <= x + 1;
endrule

rule rb (z > 20);
   y <= y + 2;
endrule
```

Parallel execution behaves like ra < rb = rb < ra

$Rule_a$ and $Rule_b$ are conflict-free if

$\forall s . \pi_a(s) \wedge \pi_b(s) \Rightarrow$ 1. $\pi_a(\delta_b(s)) \wedge \pi_b(\delta_a(s))$

2. $\delta_a(\delta_b(s)) == \delta_b(\delta_a(s))$

Parallel Execution can also be understood in terms of a composite rule

```
rule ra_rb((z>10)&&(z>20));
   x <= x+1; y <= y+2;
endrule
```

Executing Multiple Rules Per Cycle: *Sequentially Composable rules*

```
rule ra (z > 10);
   x <= y + 1;
endrule

rule rb (z > 20);
   y <= y + 2;
endrule
```

Parallel execution behaves like ra < rb

$Rule_a$ and $Rule_b$ are sequentially composable if

$$\forall s \,.\, \pi_a(s) \wedge \pi_b(s) \Rightarrow \pi_b(\delta_a(s))$$

Parallel Execution can also be understood in terms of a composite rule

```
rule ra_rb((z>10)&&(z>20));
   x <= y+1; y <= y+2;
endrule
```


Sequentially Composable rules ...

```
rule ra (z > 10);
   x <= 1;
endrule

rule rb (z > 20);
   x <= 2;
endrule
```

Parallel execution can behave either like ra < rb or rb < ra but the two behaviors are not the same

Composite rules

Behavior ra < rb

```
rule ra_rb(z>10 && z>20);
   x <= 2;
endrule
```

Behavior rb < ra

```
rule rb_ra(z>10 && z>20);
   x <= 1;
endrule
```

Compiler determines if two rules can be executed in parallel

$Rule_a$ and $Rule_b$ are conflict-free if

$\forall s \; . \; \pi_a(s) \wedge \pi_b(s) \Rightarrow$

1. $\pi_a(\delta_b(s)) \wedge \pi_b(\delta_a(s))$
2. $\delta_a(\delta_b(s)) == \delta_b(\delta_a(s))$

$D(Ra) \cap R(Rb) = \phi$
$D(Rb) \cap R(Ra) = \phi$
$R(Ra) \cap R(Rb) = \phi$

$Rule_a$ and $Rule_b$ are sequentially composable if

$\forall s \; . \; \pi_a(s) \wedge \pi_b(s) \Rightarrow \pi_b(\delta_a(s))$

$D(\pi_b) \cap R(Ra) = \phi$

These properties can be determined by examining the domains and ranges of the rules in a pairwise manner.

These conditions are sufficient but not necessary.
Parallel execution of CF and SC rules does not increase the critical path delay

Mutually Exclusive Rules

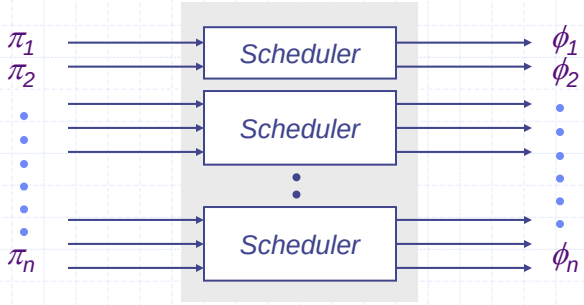
- $Rule_a$ and $Rule_b$ are mutually exclusive if they can never be enabled simultaneously

$$\forall s \; . \; \pi_a(s) \Rightarrow \sim \pi_b(s)$$

Mutually-exclusive rules are Conflict-free even if they write the same state

Mutual-exclusive analysis brings down the cost of conflict-free analysis

Multiple-Rules-per-Cycle Scheduler



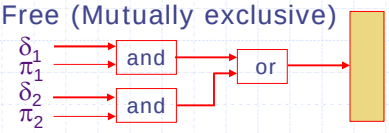
Divide the rules into smallest conflicting groups; provide a scheduler for each group

1. $\phi_i \Rightarrow \pi_i$
2. $\pi_1 \vee \pi_2 \vee .... \vee \pi_n \Rightarrow \phi_1 \vee \phi_2 \vee .... \vee \phi_n$
3. *Multiple operations such that*
   $\phi_i \wedge \phi_j \Rightarrow$ *R_i and R_j are conflict-free or sequentially composable*

Muxing structure

- Muxing logic requires determining for each register (action method) the rules that update it and under what conditions

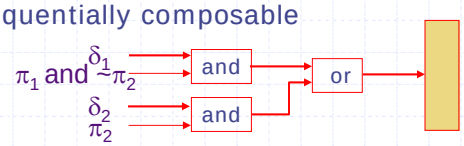
Conflict Free (Mutually exclusive)



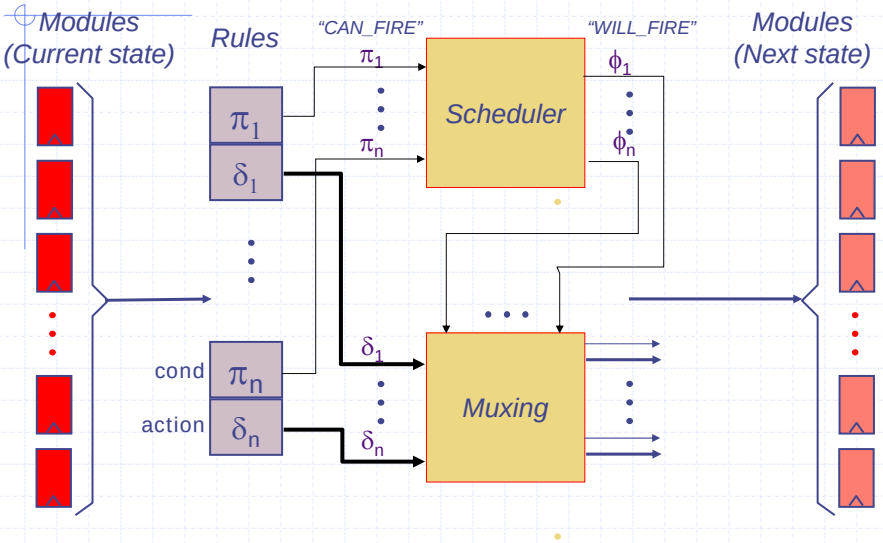
CF rules either do not update the same element or are ME

$\pi_1 \rightarrow \sim\pi_2$

Sequentially composable

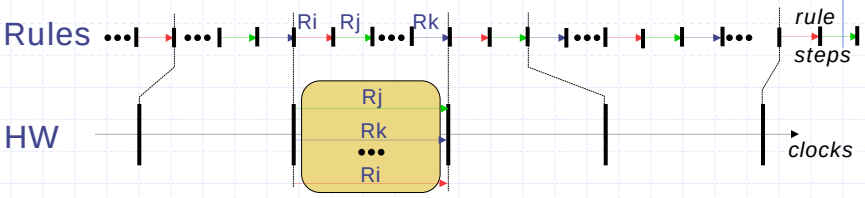


Scheduling and control logic



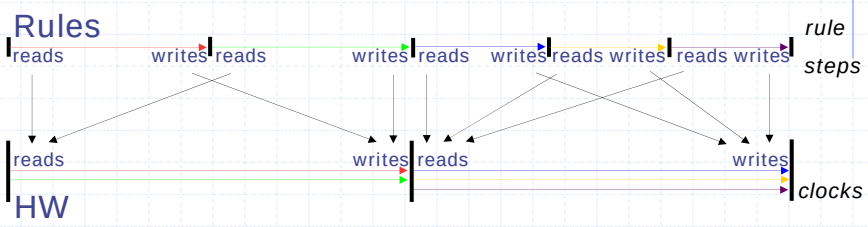
some insight

Pictorially



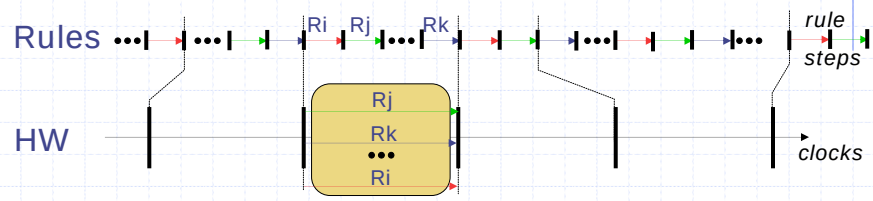
- There are more intermediate states in the rule semantics (a state after each rule step)
- In the HW, states change only at clock edges

Parallel execution reorders reads and writes



- In the rule semantics, each rule sees (reads) the effects (writes) of previous rules
- In the HW, rules only see the effects from previous clocks, and only affect subsequent clocks

Correctness

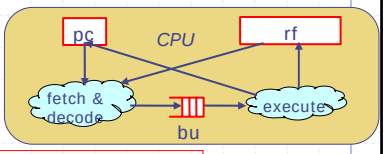


- Rules are allowed to fire in parallel only if the net state change is equivalent to sequential rule execution (i.e., CF or SC)
- Consequence: the HW can never reach a state unexpected in the rule semantics

Synthesis Summary

- Bluespec generates a *combinational hardware scheduler* allowing multiple enabled rules to execute in the same clock cycle
  - The hardware makes a rule-execution decision on every clock (i.e., it is not a static schedule)
  - Among those rules that CAN_FIRE, only a subset WILL_FIRE that is consistent with a Rule order
- Since multiple rules can write to a common piece of state, the compiler introduces appropriate muxing logic
- For proper pipelining, dead-cycle elimination and value forwarding, the user needs some understanding and control of scheduling

Two-stage Pipeline

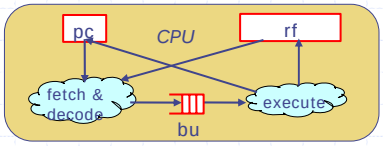



```
rule fetch_and_decode (!stallfunc(instr, bu));
        bu.enq(newIt(instr,rf));
        pc <= predIa;
endrule
```

Can these rules fire concurrently ?

```
rule execute (True);
  case (it) matches
    tagged EAdd{dst:.rd,src1:.va,src2:.vb}: begin
          rf.upd(rd, va+vb); bu.deq(); end
    tagged EBz {cond:.cv,addr:.av}:
          if (cv == 0) then begin
              pc <= av; bu.clear(); end
          else bu.deq();
    tagged ELoad{dst:.rd,addr:.av}: begin
          rf.upd(rd, dMem.read(av)); bu.deq(); end
    tagged EStore{value:.vv,addr:.av}: begin
          dMem.write(av, vv); bu.deq(); end
  endcase endrule
```

Two-stage Pipeline Analysis



1. fetch < execute

2. execute < fetch

Scheduling expectations:

execute < fetch schedule

```
rule fetch_and_decode (!stallfunc(instr, bu));
        bu.enq(newIt(instr,rf));
        pc <= predIa;
endrule
```

```
rule execute (True);
  case (it) matches
    tagged EAdd{dst:.rd,src1:.va,src2:.vb}: begin
         rf.upd(rd, va+vb); bu.deq(); en
    tagged EBz {cond:.cv,addr:.av}:
         if (cv == 0) then begin
             pc <= av; bu.clear(); end
         else bu.deq();
    tagged ELoad{dst:.rd,addr:.av}: begin
         rf.upd(rd, dMem.read(av)); bu.deq();
    tagged EStore{value:.vv,addr:.av}: begin
         dMem.write(av, vv); bu.deq(); end
  endcase endrule
```

pc:

bu:

rf:

One Element FIFO Analysis

```
module mkFIFO1 (FIFO#(t));
  Reg#(t)    data  <- mkRegU();
  Reg#(Bool) full  <- mkReg(False);
  method Action enq(t x) if (!full);
    full <= True;     data <= x;
  endmethod
  method Action deq() if (full);
    full <= False;
  endmethod
  method t first() if (full);
    return (data);
  endmethod
  method Action clear();
    full <= False;
  endmethod
endmodule
```

deq < enq ?

first < deq ?

first < enq ?

Expectation bu: (first<deq) < (find<enq)

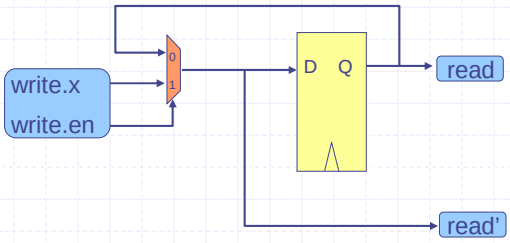
The good news ...

- It is always possible to transform your design to meet desired concurrency and functionality

Register Interfaces

read < write

write < read ?

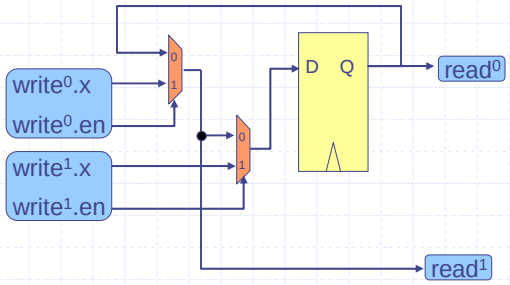


read' – returns the current state when write *is not enabled*
read' – returns the value being written if write *is enabled*

Ephemeral History Register (EHR)

[MEMOCODE'04]

$read^0 < write^0 < read^1 < write^1 < ....$



$write^{i+1}$ takes precedence over $write^i$

Transformation for Performance

```
rule fetch_and_decode (!stallfunc(instr, bu)[1]);
        bu.enq[1](newIt(instr,rf));
        pc <= predIa;
endrule
```

```
rule execute (True);
  case (it) matches
    tagged EAdd{dst:.rd,src1:.va,src2:.vb}: begin
         rf.upd[0](rd, va+vb); bu.deq[0](); end
    tagged EBz {cond:.cv,addr:.av}:
         if (cv == 0) then begin
              pc <= av; bu.clear[0](); end
         else bu.deq[0]();
    tagged ELoad{dst:.rd,addr:.av}: begin
         rf.upd[0](rd, dMem.read(av)); bu.deq[0](); end
    tagged EStore{value:.vv,addr:.av}: begin
         dMem.write(av, vv); bu.deq[0](); end
  endcase endrule
```


One Element FIFO *using EHRs*

```
module mkFIFO1 (FIFO#(t));                    first[0] < deq[0] < enq[1]
  EHReg2#(t)    data  <- mkEHReg2U();
  EHReg2#(Bool) full  <- mkEHReg2(False);
  method Action enq[0](t x) if (!full.read[0]);
    full.write[0] <= True;  data.write[0] <= x;
  endmethod
  method Action deq[0]() if (full.read[0]);
    full.write[0] <= False;
  endmethod
  method t first[0]() if (full.read[0]);
    return (data.read[0]);
  endmethod
  method Action clear[0]();
    full.write[0] <= False;
  endmethod
endmodule
```

After Renaming

- Things will work
  - both rules can fire concurrently

Programmer Specifies:

$$R_{execute} < R_{fetch}$$

Compiler Derives:

$$(first^0, deq^0) < (find^1, deq^1)$$

What if the programmer wrote this?

$$R_{execute} < R_{execute} < R_{fetch} < R_{fetch}$$

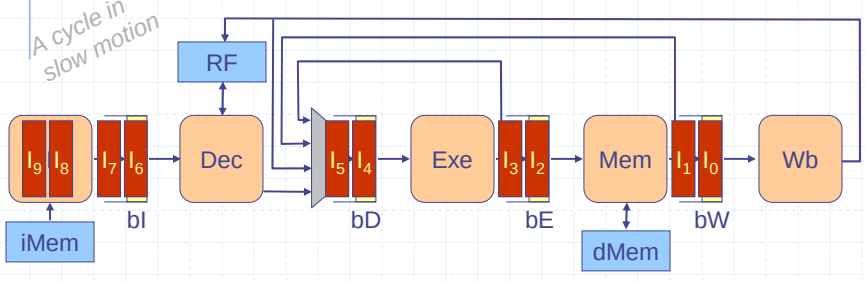
Experiments in scheduling

Dan Rosenband, ICCAD 2005

- What happens if the user specifies:

Wb < Wb < Mem < Mem < Exe < Exe < Dec < Dec < IF < IF

No change in rules *a superscalar processor!*



Executing 2 instructions per cycle requires more resources but is functionally equivalent to the original design